**ASCII Art Generator: From Image to Text Art**

Code link:

<https://colab.research.google.com/drive/17XW-2Qa2dqkIMQ4zoknRA-Zt9G1OUTFm#scrollTo=VbQccDjR8L4e>

**1. Problem Domain and Project Description**

The aim of this project is to create an ASCII Art Generator that takes an image as input and produces an ASCII text version of the image as output. So, basically, it converts the brightness levels of each pixel in an image into corresponding ASCII characters like ‘@’, ‘#’, ‘.’, and spaces. The input is any image i.e. a .jpg or .png file, and the output is a text-based representation of that image where the characters represent varying shades of brightness. For instance, if I use an image of a famous painting, the output would be a textual rendition that somewhat resembles the painting, but made entirely out of ASCII characters. It’s a cool way to give a retro, artistic feel to any image, especially in low-resolution or low-bandwidth environments.

**2. Detailed Approach**

So, let me tell you how I approached this. It was broken down into a few steps:

**Loading the Image:** First, I had to read the image file using Python’s PIL library. This library handles all the image manipulation tasks like opening, resizing, and converting the image into grayscale.

**Resizing the Image:** The next step was to resize the image, but it’s not just as simple as scaling down. ASCII characters don’t have the same aspect ratio as pixels, so I needed to adjust the height to make sure the final ASCII art doesn’t look squished or stretched. I used an aspect ratio adjustment (specifically a factor of 0.55) to ensure it looks right when printed as text.

**Converting to Grayscale:** Once resized, I converted the image into grayscale. This is where each pixel in the image gets assigned a single intensity value, from 0 (black) to 255 (white). This makes it easy to map brightness levels to characters later.

**Mapping Pixels to ASCII Characters:** Here’s where the magic happens! Each pixel’s brightness is mapped to a corresponding ASCII character. Darker pixels get denser characters like @ or #, while lighter pixels get more sparse characters like ‘.’ or spaces. I used a predefined set of characters to represent these brightness levels.

**Generating ASCII Art:** Once every pixel has been mapped to a character, I created a big string where each character corresponds to a pixel. This is printed to the console, or you can save it as a text file for viewing later.

**Challenges and Data Handling:**

The actual complexity of this approach was not too high, but there were some challenges in balancing image resizing and character mapping to maintain the visual integrity of the image. For example, if the image was too detailed, the ASCII art would lose its structure when converted to text. So, adjusting the resolution and character set played a crucial role.

I didn’t face any data problems since the project only required simple images as input. The task didn’t demand a large dataset or any special image preprocessing steps. I could work with any image format (like .jpg, .png), and that made it versatile.

**3. Results and Comparisons to Alternatives**

The results are really interesting! The generator successfully converts images into ASCII representations that look surprisingly close to the originals. Of course, the final ASCII art is more abstract than an actual image, but if you squint or step back a bit, you can easily recognize the shapes and details.

Compared to other techniques like pixelation or low-resolution image compression, this method is much more artistic and playful. While pixelation tries to preserve the detail in the image, ASCII art focuses on creating a creative, stylized version of the image using only characters.

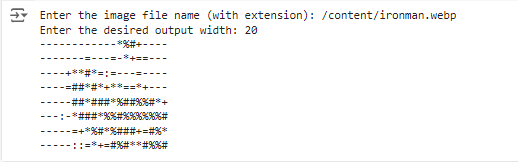
**Does This Solve the Problem?**

I’d say yes, it definitely solves the problem of converting images into an artistic ASCII format. It’s great for low-bandwidth environments or situations where images can’t be easily displayed (like a terminal window). It’s not meant for precision or detailed image manipulation, but for what it sets out to do—convert an image into a fun, text-based representation—it works perfectly.

1. Iron Man



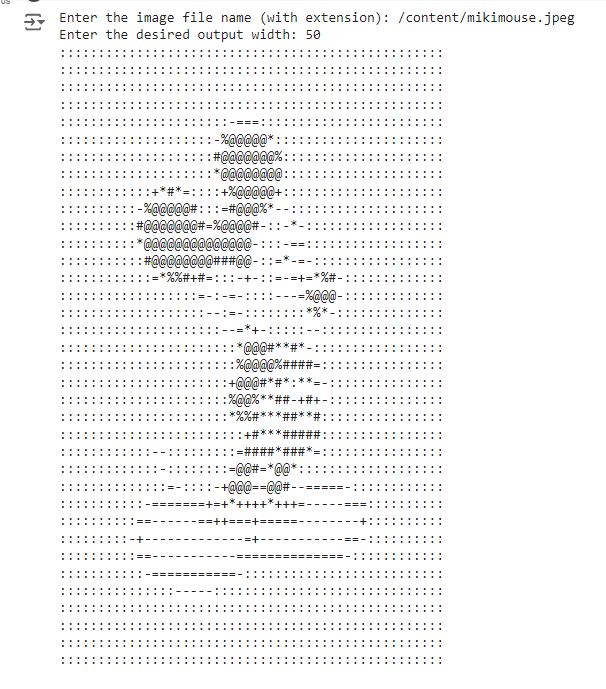
Width 100px Width 50px



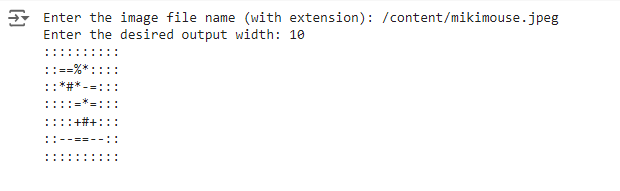
Width 10px

1. Miki Mouse:





Width 100px Width 50px



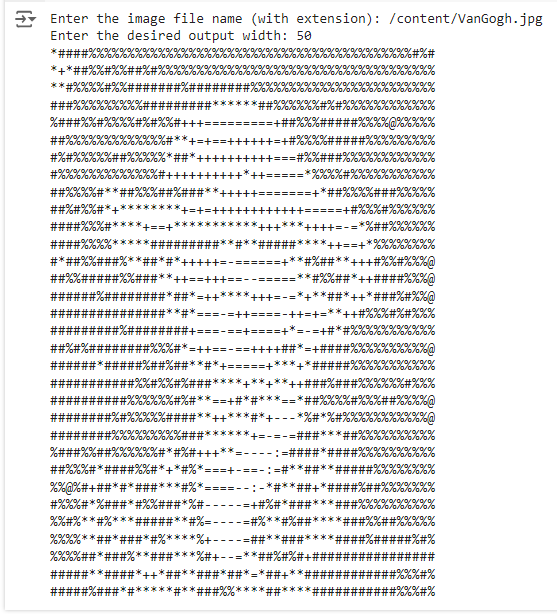
Width 10px

1. Van Gogh:

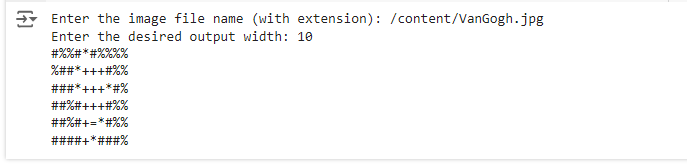




Width 120px



Width 50px



Width 10px

**Results and Analysis**

The results are really interesting! The generator successfully converts images into ASCII representations that look surprisingly close to the originals.

When the images (Iron Man, Mickey Mouse, and Van Gogh) were converted to ASCII art with different widths, the results were significantly influenced by the output size:

**Width 100 or 120px:** The images were highly detailed and closely resembled the original. The larger width allowed for a finer representation of the pixel values, resulting in more accurate mappings to the ASCII characters. The nuances in shading and shapes were preserved, making the ASCII version recognizable.

**Width 50px:** At this width, the images still retained their basic structure but lost some of the finer details. For example, in complex images like Van Gogh’s painting, intricate textures might become less discernible. However, major shapes and contrasts (like the outline of Iron Man’s mask) were still distinguishable.

**Width 10px:** At this small width, the output was highly abstract and difficult to recognize. Due to the limited number of characters per row, the system had to compress significant amounts of visual information, which led to the loss of recognizable features. For example, the Mickey Mouse image at a width of 10px only vaguely resembles a round shape, but most details, including facial features, would be entirely lost. The reason for this is that the reduced resolution can no longer capture the complexity of the original image, and the limited ASCII character set doesn't have enough variation to make up for the loss of detail.

The width of the ASCII output directly affects how much of the original image's detail can be preserved. At wider widths (100), more characters are used per line, allowing the system to represent more brightness variations across a larger number of pixels. This results in more accurate and recognizable ASCII art. However, at smaller widths (like 10), there are fewer characters available, meaning the system has to condense larger areas of the image into single characters. This compression causes a significant loss of detail, leading to abstract or unrecognizable outputs.

Compared to other techniques like pixelation or low-resolution image compression, this method is much more artistic and playful. While pixelation tries to preserve the detail in the image, ASCII art focuses on creating a creative, stylized version of the image using only characters.

**4. Tools and Resources I Used**

Here’s a quick rundown of the tools and libraries that helped me complete the project:

Python: The programming language I used to implement the entire logic.

Pillow (PIL): This library helped with all the image manipulation, like resizing, opening, and converting images to grayscale.

NumPy: I used NumPy to handle pixel data efficiently. It’s great for converting the image data into arrays for easier manipulation.

**Inspirations and Resources:**

<https://pillow.readthedocs.io/en/stable/>

<https://github.com/TheZoraiz/ascii-image-converter>

**Conclusion**

In summary, this ASCII Art Generator was a fun project that transformed regular images into unique text-based art. By manipulating image size, brightness, and ASCII characters, I was able to create stylized outputs where large pixel pictures look surprisingly close to the original images. While this method is more about artistic expression than precision, it certainly succeeds in converting images into text-based representations, perfect for retro computing fans or low-bandwidth applications.